Chapitre 5

Maintenant que nous savons comment appeler des ROM_CALLs prenant en paramètres des entiers, ou une chaîne de caractère, nous allons apprendre à optimiser ces appels ; nous en profiterons pour montrer qu'utiliser des ROM_CALLs prenant plusieurs types d'arguments différents n'est pas plus difficile que ce que nous avons jusqu'à présent étudié.

I:\ Appel d'un ROM_CALL avec des paramètres de types différents

Nous allons en premier lieu voir comment appeler le ROM_CALL `DrawStr`, qui vous sera probablement souvent utile, puisqu'il permet un affichage de texte à l'écran, à la position, exprimée en pixels, que vous désirez.
Comme nous l'indique la documentation de TIGCC, voici comment ce ROM_CALL est déclaré :

```
void DrawStr (short x, short y, const char *str, short Attr);
```

Le premier paramètre doit être un entier, codé sur deux octets, puisqu'il s'agit d'un short, et correspond à la position en abscisse à laquelle le message sera exprimé à l'écran. Comme pour `DrawLine`, que nous avons étudié au chapitre précédent, la position est exprimée en nombre de pixels par rapport au coin supérieur gauche de l'écran (Le pixel tout dans le coin haut gauche de la machine a pour coordonnées en abscisse 0, et même chose pour la coordonnée en ordonnée).
Le second paramètre est de même type, mais désigne la position en ordonnée, c'est à dire la ligne (toujours en pixels), ou sera affiché le message.
Le troisième paramètre est de même type que celui attendu par le ROM_CALL `ST_helpMsg`, que nous avons aussi utilisé pour exemple au chapitre précédent. Il désigne la chaîne de caractères que nous souhaitons afficher à l'écran, et s'utilise exactement comme vu plus tôt.
Enfin, le quatrième et dernier paramètre est un entier, comme les deux premiers, et peux prendre cinq valeurs différentes, selon la façon dont nous voulons afficher notre message ; ci-dessous, un tableau de correspondance entre les valeurs et les modes d'affichage (à titre d'information, puisque c'est ce qui est utilisé dans la documentation de TIGCC, nous indiquons, pour chaque valeur, le code correspondant utilisé en langage C ; nous n'en tiendrons pas compte ici, mais vous saurez à quelles valeurs ils correspondent, si vous les "croisez" dans la documentation):

ASM	C	Résultat
0	A_REVERSE	Le texte est affiché en blanc sur fond noir.
1	A_NORMAL	Le texte est affiché en noir sur fond blanc.
2	A_XOR	Le texte est dessiné en couleurs inversant le fond (là où le texte est noir et le fond aussi, ce sera affiché en blanc, par exemple : c'est un OU exclusif).
3	A_SHADED	Le texte est dessiné en "ombré" ; seuls la moitié des pixels le constituant sont affichés.
4	A_REPLACE	Le texte est écrit en effaçant l'arrière plan.

Pour ce ROM_CALL comme pour tous les autres, il convient de placer les paramètres sur la pile, en ordre inverse de l'écriture C.
Nous allons afficher les texte "Salut !" à partir du pixel dont le couple de coordonnées est [10;25], en mode normal (qui a pour code 1), ce qui correspond à une écriture en noir sur blanc. Pour laisser à l'utilisateur du programme le temps de lire le message, nous attendrons un appui sur une touche à l'aide du ROM_CALL `ngetchx` avant de rendre le contrôle au TIOS.
Pour cela, nous agirons comme nous l'avons déjà fait au chapitre précédent : la seule différence est que, ici, nous alternons les types de paramètres. C'est pour cela que nous ne fournirons pas d'explications détaillées sur notre exemple, que vous trouverez ci-dessous. Si vous ne comprenez pas exactement ce qu'il fait, nous vous conseillons de réétudier le chapitre précédent de ce tutorial, car comprendre, et maîtriser les appels de ROM_CALL, et donc, le passage de paramètres, est indispensable à la programmation en langage d'Assembleur, du moins pour l'usage que nous ferons de ce langage dans ce tutorial.
(Si, une fois la lecture de ce langage terminée, vous continuez pendant pas mal de temps la programmation en ASM, vous écrirez grand nombre de fonctions, qui s'appellent généralement de la même façon que les ROM_CALL, et vous aurez encore besoin de ce que nous apprenons ici.)

Voici le code source du programme permettant d'afficher notre petit message :

```
; Assembly Source File
; Created 11/02/2003, 23:18:07

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    move.w  #1,-(a7)          ; Mode d'affichage
    pea.l   texte(pc)         ; Le message qu'on veut afficher
    move.w  #25,-(a7)         ; Ordonnée
    move.w  #10,-(a7)         ; Abscisse
    move.l  $C8,a0
    move.l  DrawStr*4(a0),a0
    jsr     (a0)
    lea     10(a7),a7

    move.l  $C8,a0
    move.l  ngetchx*4(a0),a0
    jsr     (a0)

    rts

texte:      dc.b    "Salut !",0
```

Exemple Complet

II:\ Une petite optimisation... pas si petite que cela !

Comme nous l'avons déjà souligné à la fin du chapitre précédent, et comme vous avez de nouveau pu le constater dans l'exemple que nous venons de prendre, à chaque fois que nous avons appelé un ROM_CALL, nous avons du au préalable exécuter l'instruction suivante :

```
move.l  $C8,a0
```

Cela n'est certes pas gênant pour deux ROM_CALLs, mais cela le deviendra pour un plus grand nombre : d'une, cela prend du temps à écrire, de deux, ça augmente la taille du programme, et, de trois, cela ralenti le programme (puisque chaque instruction prend de la place en mémoire et nécessite un petit temps pour s'exécuter).
L'optimisation que nous allons ici présenter sera plus une pessimisation qu'autre chose pour un exemple aussi bref, mais elle sera absolument non négligeable pour des programmes un peu plus long ; je vous conseille donc de l'utiliser aussi souvent que possible (Selon toute logique, vous ne vous limiterez pas à des programmes ne comportant que deux ou trois ROM_CALL !).

L'optimisation est simple : l'astuce est de mémoriser $C8 dans un registre (comme nous le faisions à présent, dans a0), mais pas a0. Nous allons maintenant voir pourquoi... et quelles sont les opérations à effectuer en plus de cela, pour laisser la calculatrice dans un état stable une fois l'exécution du programme terminée.

A: Registres modifiables... et registres modifiés

Registres que le prog a le droit de modifier
=> Les RC ont le droit de modifier les mêmes
=> On ne peut pas stocker d'information dans ces registres entre plusieurs appels de RC
=> Il va falloir en utiliser un autre

B: Sauvegarde, puis restauration de registres

Comment faire pour sauvegarder un registre.
Comment faire pour le restaurer

C: Enfin, l'optimisation

A présent, nous pouvons voir l'optimisation effective.

Pour stocker $C8, il nous faut, comme a0, un registre d'adresse. Puisque nous ne pouvons pas utiliser les registres de bas numéros, qui peuvent être effacés par les ROM_CALL, ni le registre a7, qui est le pointeur de pile (le modifier sans faire grandement attention entraînera un plantage quasi-immédiat !), il nous faut en choisir un de numéro "intermédiaire".
Nous pourrions choisir d'utiliser le registre a6... mais il est souvent utilisé pour autre chose, que ce soit en C ou en ASM...
Donc, par convention, si l'on peut aller jusqu'à dire cela, nous utiliserons le registre a5. Ce parce que c'est celui-ci qui est utilisé par TIGCC lorsque l'on programme en C, et parce que c'est celui que les programmeurs ASM ont l'habitude d'employer. En utilisant ce registre, vous rendrez votre programme plus facilement compréhensible par d'autres programmeurs.

```
; Assembly Source File
; Created 11/02/2003, 23:18:07

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    move.l  a5,-(a7)          ; Sauvegarde du registre a5

    move.l  $C8,a5

    move.w  #1,-(a7)          ; Mode d'affichage
    pea.l   texte(pc)         ; Le message qu'on veut afficher
    move.w  #25,-(a7)         ; Ordonnée
    move.w  #10,-(a7)         ; Abscisse
    move.l  DrawStr*4(a5),a0
    jsr     (a0)
    lea     10(a7),a7

    move.l  ngetchx*4(a5),a0
    jsr     (a0)

    move.l  (a7)+,a5          ; Restauration du registre a5

    rts

texte:    dc.b    "Salut !",0
```

Exemple Complet